Makroprogrammierung in LaTeX

Marei Peischl (marei@peiTeX.de)

2022-03-11

Workshop im Rahmen Chemnitzer Linuxtage 2023

Wieso seid ihr hier?

Wieso sollte ich in LaTeX programmieren?

Wieso sollte ich in LaTeX programmieren?

Weil es geht!

Wieso sollte ich in LaTeX programmieren?

Weil es geht!

Automatisierung – Effizienz – Flexibilität

Auffrischung

Auffrischung: Standarddefinitionen

Makrostruktur

```
\Befehl
\BefehlMitArgument*[optionales Argument]{Argument}
\begin{Umgebung}...\end{Umgebung}
```

Auffrischung: Standarddefinitionen

Makrostruktur

```
\Befehl
\BefehlMitArgument*[optionales Argument]{Argument}
\begin{Umgebung}...\end{Umgebung}
```

Makrodefinitionen

```
\newcommand*{\Befehlsname}{Definition},
\newenvironment*{Umgebungsname}[Anzahl][Säumniswert]{Start-Code}
    {Ende-Code}
```

Grenzen der Standarddefinitionen

- maximal 9 Parameter
- optionale immer vor den notwendigen Argumenten
- Abhängigkeit der Argumente untereinander
- keine anderen Begrenzer als `{}[]`

Makrodefinitionen 2.0 – xparse

Das Paket xparse (Muss nicht mehr geladen werden)

Basis für xparse ist expl3

- Entwicklung noch nicht vollständig abgeschlossen
- gibt auch experimentelle Anweisungen
- hier nur die stabilen Möglichkeiten

Neue Möglichkeiten der Makrodefinition

```
\DeclareDocumentCommand{\Makro}{Argumentspezifikation}{Code}
\NewDocumentCommand{\Makro}{Argumentspezifikation}{Code}
\RenewDocumentCommand{\Makro}{Argumentspezifikation}{Code}
\ProvideDocumentCommand{\Makro}{Argumentspezifikation}{Code}
```

Notwendige Argumente

`m`	Standardargument. Token oder Gruppe

Notwendige Argumente

`m`	Standardargument. Token oder Gruppe
`r`*Token1Token2*	Begrenztes argument, z. B. `r<>`. Fehlt das öffnende Token wird ein Fehler generiert und stattdessen `\NoValue` übergeben

Notwendige Argumente

`m`	Standardargument. Token oder Gruppe
`r` *Token1Token2*	Begrenztes argument, z. B. `r<>`. Fehlt das öffnende Token wird ein Fehler generiert und stattdessen `\NoValue` übergeben
`R` *Token1Token2* **`{`** *default* **`}`**	Wie `r`, allerdings *default* anstelle von `\NoValue`

Notwendige Argumente

`m`	Standardargument. Token oder Gruppe
`r`*Token1Token2*	Begrenztes argument, z. B. `r<>`. Fehlt das öffnende Token wird ein Fehler generiert und stattdessen `\NoValue` übergeben
`R`*Token1Token2*`{`*default*`}`	Wie `r`, allerdings *default* anstelle von `\NoValue`
`v`*Token1Token2*	Wie r, aber „verbatim“. In Argumenten nicht zulässig.

Notwendige Argumente

m	Standardargument. Token oder Gruppe
r*Token1Token2*	Begrenztes argument, z. B. r<>. Fehlt das öffnende Token wird ein Fehler generiert und stattdessen \NoValue übergeben
R*Token1Token2***{***default***}**	Wie r, allerdings *default* anstelle von \NoValue
v*Token1Token2*	Wie r, aber „verbatim“. In Argumenten nicht zulässig.
b	Inhalt einer Umgebung.

Optionale Argumente

o	optionales Standardargument in eckigen Klammern. `\NoValue` ist Defaultwert

Optionale Argumente

`o`	optionales Standardargument in eckigen Klammern. `\NoValue` ist Defaultwert
`O{`*default*`}`	Wie `o` mit anderem Defaultwert..

Optionale Argumente

`o`	optionales Standardargument in eckigen Klammern. `\NoValue` ist Defaultwert
`O{`*default***`}`**	Wie o mit anderem Defaultwert..
`d`*Token1Token2*	Wie o mit anderen Begrenzern.

Optionale Argumente

o	optionales Standardargument in eckigen Klammern. `\NoValue` ist Defaultwert
O{*default***}**	Wie o mit anderem Defaultwert..
d*Token1Token2*	Wie o mit anderen Begrenzern.
D*Token1Token2***{***default***}**	Wie d mit eingestelltem Default.

Optionale Argumente

o	optionales Standardargument in eckigen Klammern. `\NoValue` ist Defaultwert
O{*default***}**	Wie o mit anderem Defaultwert..
d*Token1Token2*	Wie o mit anderen Begrenzern.
D*Token1Token2***{***default***}**	Wie d mit eingestelltem Default.
s	Testet, ob Stern. Liefert `\BooleanTrue` oder `\BooleanFalse`.

Optionale Argumente

`o`	optionales Standardargument in eckigen Klammern. `\NoValue` ist Defaultwert
`O{`*default***`}`**	Wie `o` mit anderem Defaultwert..
`d`*Token1Token2*	Wie `o` mit anderen Begrenzern.
`D`*Token1Token2***`{`***default***`}`**	Wie `d` mit eingestelltem Default.
`s`	Testet, ob Stern. Liefert `\BooleanTrue` oder `\BooleanFalse`.
`t`*Token*	Basis von `s`; optionales Token ist frei bestimmbar .

Implementierte Tests

```
\IfNoValueTF{Argument}{Wahr}{Falsch}%optionale Argumente
\IfBooleanTF{Argument}{Wahr}{Falsch}%Sternchen
```

Einzweigige Varianten

```
\IfNoValueT{Argument}{Wahr}
\IfBooleanF{Argument}{Falsch}
```

Übung

Erstellt ein Makro, dass ein Sternchenargument, und ein optionales Argument mit einem anderen Begrenzern als `[]` erwartet.

Kontrollstrukturen mit ifthen

Kontrollstruturen bei TEX

```
\ifdim\linewidth<15cm
    Falls kleiner als 15 cm
\else
    Falls größer oder gleich 15 cm
\fi
```

Das Standard-LaTeX2e-Paket für Konstrollstrukturen

Bedingte Verzweigung

\ifthenelse{*Test*}{*wenn Test wahr*}{*wenn Test falsch*}

Das Standard-LaTeX2e-Paket für Konstrollstrukturen

Bedingte Verzweigung

`\ifthenelse`{*Test*}{*wenn Test wahr*}{*wenn Test falsch*}

Schleifen

`\whiledo`{*Test*}{*Code, der wiederholt ausgeführt werden soll, wenn Test wahr*}

Mögliche Tests i

Wahrheitswerte

`\boolean{`*Name*`}`

Existenz von Makros

`\isundefined{`*Makroname*`}`

Zeichenkettenvergleich

`\equal{`*Zeichenkette1*`}{`*Zeichenkette2*`}`

Mögliche Tests ii

Ganzzahlen

Zahl<*Zahl*

Zahl>*Zahl*

Zahl=*Zahl*

`\isodd{`*Zahl*`}`

Längen

`\lengthtest{`*Länge1* < *Länge2*`}`

`\lengthtest{`*Länge1* > *Länge2*`}`

`\lengthtest{`*Länge1* = *Länge2*`}`

Kombination verschiedener Tests

```
\AND
\OR
\NOT
\( \)
```

Booleans

```
\newboolean{Booleanname}
\provideboolean{Booleanname}
\setboolean{Booleanname}{true/false}
```

Übung

Schreibt ein Makro, dass ein ganzzahliges Argument verarbeitet und von diesem an bis 0 zählt. Für `\countdown{10}` wird beispielsweise folgende Ausgabe erzeugt:

10–9–8–7–6–5–4–3–2–1–0

Übung

Schreibt ein Makro, dass ein ganzzahliges Argument verarbeitet und von diesem an bis 0 zählt. Für `\countdown{10}` wird beispielsweise folgende Ausgabe erzeugt:

10–9–8–7–6–5–4–3–2–1–0

Auffrischung: Zähler

\newcounter{*Zählername*}

\stepcounter{*Zählername*}

Expansionskontrolle

Befehl wird durch seine Bedeutung ersetzt

```
\newcommand*{\Variable}{def}
\meaning\Variable\\
\newcommand*{\Funktion}[1]{Funktion mit Argument (#1)}
\meaning\Funktion
```

Befehl wird durch seine Bedeutung ersetzt

```
\newcommand*{\Variable}{def}
\meaning\Variable\\
\newcommand*{\Funktion}[1]{Funktion mit Argument (#1)}
\meaning\Funktion
```

macro:->def

macro:#1->Funktion mit Argument (#1)

Einmalige Expansion

```
\newcommand\eins{eins}
```

\eins

eins

```
\newcommand\zwei{\eins, zwei}
```

\zwei

\eins ,zwei

Mehrstufige Expansion

```
\newcommand \eins {eins}
\newcommand \zwei {\eins ,zwei}
\newcommand \drei {\eins ,\zwei ,drei}
```

Mehrstufige Expansion

\newcommand \eins {eins}

\newcommand \zwei {\eins ,zwei}

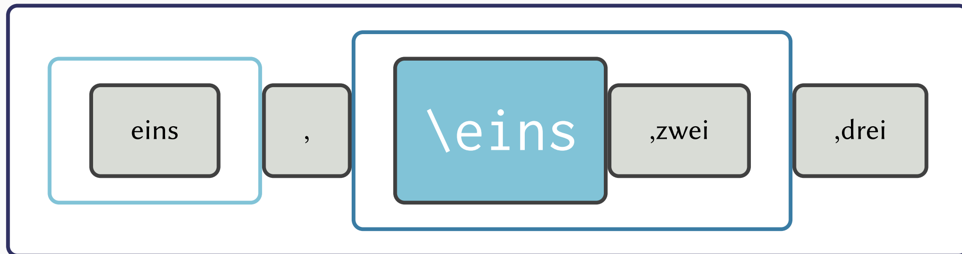
\newcommand \drei {\eins ,\zwei ,drei}

Mehrstufige Expansion

\newcommand \eins {eins}

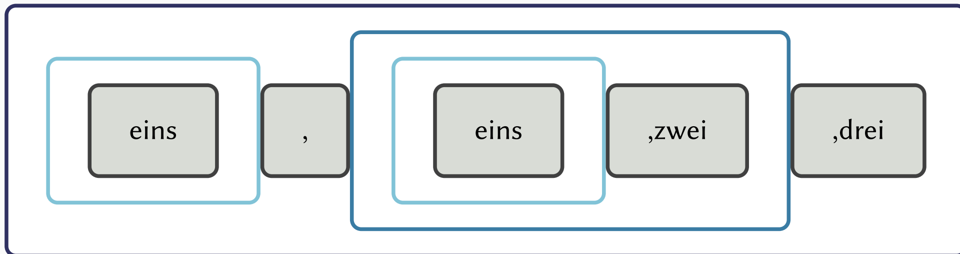
\newcommand \zwei {\eins ,zwei}

\newcommand \drei {\eins ,\zwei ,drei}



Mehrstufige Expansion

```
\newcommand \eins {eins}
\newcommand \zwei {\eins ,zwei}
\newcommand \drei {\eins ,\zwei ,drei}
```



Expansion beeinflussen

```
\noexpand
\protect
\expandafter
```

Beispiel expandafter

```
\newcommand*{\Argumente}{{eins}{zwei}}
\newcommand*{\Makro}[2]{1:#1 -- 2:#2}
\Makro\Argumente Text danach\\
\expandafter\Makro\Argumente Text danach
```

Ausgabe

1:einszwei – 2:Text danach

1:eins – 2:zweiText danach

csname/endcsname

Eigene Definitionen

```
\newcommand{\Makro}{Code}
```

Aufruf: `\Makro`

Eigene Definitionen

```
\newcommand{\Makro}{Code}
```

Aufruf: `\Makro`

```
\expandafter\newcommand\csname Makroname\endcsname{Code}
```

Aufruf: `\Makroname` oder `\csname Makroname\endcsname`

Warum sich die Mühe machen?

- Nicht definierte Kommandos bedeuten `\relax`

Warum sich die Mühe machen?

- Nicht definierte Kommandos bedeuten `\relax`
- Kommandos können andere Zeichen als Buchstaben enthalten

Warum sich die Mühe machen?

- Nicht definierte Kommandos bedeuten `\relax`
- Kommandos können andere Zeichen als Buchstaben enthalten
- Makro Argumente können innerhalb der Konstrukte verwendet werden

Warum sich die Mühe machen?

- Nicht definierte Kommandos bedeuten `\relax`
- Kommandos können andere Zeichen als Buchstaben enthalten
- Makro Argumente können innerhalb der Konstrukte verwendet werden
- Expanding innerhalb einer mit `\csname` definierten Konstruktion ist möglich

Referenzen

```
\csname name\endcsname
```

Example

```
\label{frame:csname}
\expandafter\meaning\csname r@frame:csname\endcsname
```

macro:->{26}{53}{csname/endcsname}{Doc-Start}{}

Übung

Definiert anschließend innerhalb einer Schleife eine Reihe von Makros, die Zahlenwerte enthalten, z. B. „Makro1“ usw.

Prüfen, ob ein Makro definiert ist

```
\expandafter\ifx\csname FooBar\endcsname\relax
    Wenn \string\FooBar nicht definiert ist
\else
    Wenn \FobBar definiert ist
\fi
```

Eigene Klassen und Pakete

Die Struktur von Klassen oder Paketen

1. Identifikation: Klasse oder Paket mit kurzer Beschreibung
2. Vorläufige Deklarationen: Deklarationen & Laden anderer Dateien. Lediglich soviel Code wie für die Optionsdeklaration gebraucht wird.
3. Optionen: Deklaration und Verarbeitung von Optionen
4. Weitere Deklarationen: Hier wird der Hauptteil der Datei gesetzt. Deklaration neuer Variablen, Makros und Fonts, und Laden anderer Pakete

Identifikation

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{Paketname}[Datum Weitere Infos]
\ProvidesClass{Klassenname}[Datum Weitere Infos]
\ProvidesFile{Dateiname}[Datum Weitere Infos]
```

Auf anderen Klassen und Paketen aufbauen

```
\RequirePackage[Optionen]{Paket}
\RequirePackageWithOptions{Paket}
\LoadClass[Optionen]{Klasse}
\LoadClassWithOptions{Klasse}
```

Optionen

```
\DeclareOption{Optionsname}{Code}
\DeclareOption*{Code}
```

Optionen

```
\DeclareOption{Optionsname}{Code}
\DeclareOption*{Code}
```

Optionen weitergeben

```
\PassOptionsToPackage{Optionen}{Paket}
\PassOptionsToClass{Optionen}{Klasse}
```

Optionen verarbeiten

```
\ProcessOptions*
```

Optionsweitergabe

Example

```
\DeclareOption{%
    \PassOptionsToClass{\CurrentOption}{article}%
}
```

Weitere nützliche Befehle

Code verzögern

```
\AtEndOfClass{Code}
\AtEndOfPackage{Code}
\AtBeginDocument{Code}
\AtEndDocument{Code}
```

IfFileExists

```
\IfFileExists{Dateiname}{Code wenn da}{Code wenn nicht da}
\InputIfFileExists{Dateiname}{Code wenn da}{Code wenn nicht da}
```

Fehlermeldungen

```
\ClassError{Klasse}{Fehlermeldung}{Hilfestellung}
\PackageError{Paket}{Fehlermeldung}{Hilfestellung}
```

Das @-Zeichen

Die Sonderrolle des @

1. Zeichen der Klasse 12: „other“
2. In `.cls`- und `.sty`-Files Zeichen der Klasse 11: „letter“

⇒ `@` in vielen internen Kommandos enthalten

```
\makeatletter
\makeatother
```

Anhang/Zusatzinfos

`\Befeh{Argument}`

`\emph{Hervorherbung}` → *Hervorhebung*

Geschweifte Klammern markieren notwendige Argumente.

Weglassen ⇒ Fehlermeldung bzw. erstes Token wird als Argument gelesen.

`\`*Befehl*`[`*optionales Argument*`]`

`$\sqrt[3]{2}$` → $\sqrt[3]{2}$

Eckige Klammern sind i. d. R. Optionen

weglassen möglich: `$\sqrt{2}$` → $\sqrt{2}$

\begin{*Umgebung*}...\end{*Umgebungsnae*}

`\begin{center} zentrierter Absatz \end{center}` →

zentrierter Absatz

Begrenzung auf Umgebungsinhalt

Argumente nur am Anfang:

\begin{*Umgebung*}[*Option*]{*Argument*}

Fehlende Zeichen in Schriftart sollen eine Fehlermeldung geben

```
\tracinglostchars=3
```

Mehrseitige Floats

hvfloats-Paket